**EXCEPTION HANDLING**

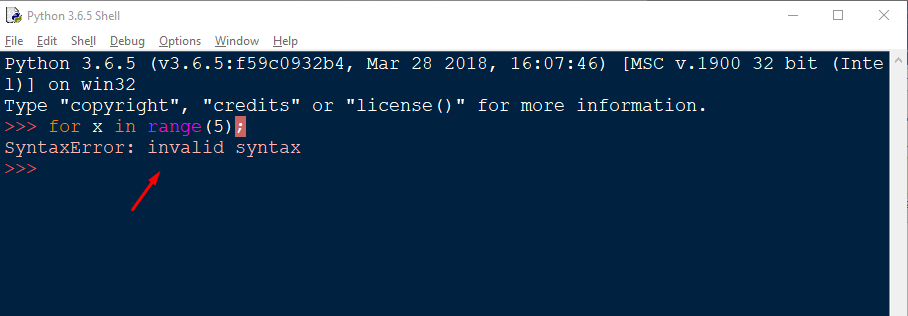
**Introduction:**

So far in python while creating your programs you might have encountered a

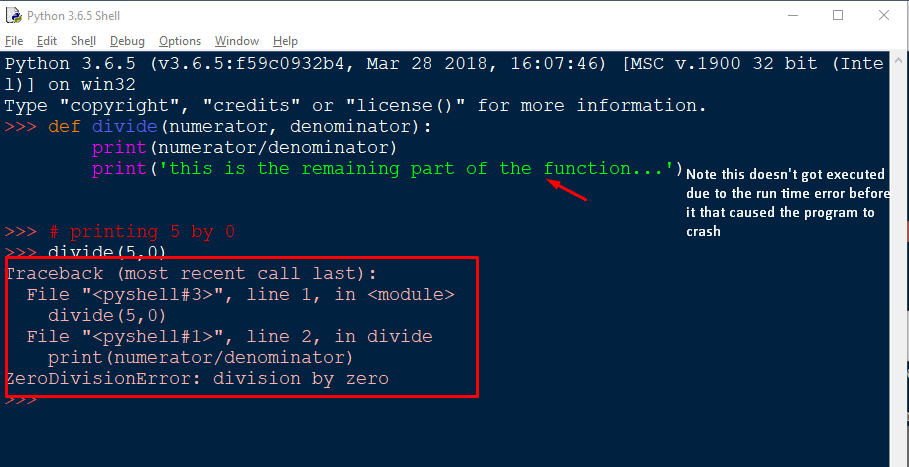
Number of errors which is not necessarily a bad thing. You may have observed it to be of two types.

1. Errors due to incorrect syntax – Syntactical Errors
2. Errors during execution – Exceptions

While the former category of errors occurred when you were writing the code itself indicated by a error message something like in the image show below.



The latter category, Exceptions occurs during the execution of program, when you are running your program. These cause your program to crash/stop abruptly like shown below in the image



Here comes the exception handling to the rescue. By using exception handling you can handle these exceptions ( run time errors ) so that your program doesn’t close abruptly/crashes and the remaining part of your code gets executed.

**EXCEPTION:**

These are the run time errors. Python has several built-in exceptions that forces the program to output the error when something goes wrong. When such an exception occurs, it causes the current process to stop (breaks the current flow of the program) and passes the exception to the calling process to handle. If it’s not handled the program will crash.

If the exception is not handled in the entire program, an error message is flashed on the screen and the program is stopped at the very point where the exception occurred preventing the further part of the program to be executed ever.

**EXCEPTION HANDLING**

Exceptions in python can be handled using the **Try-Except** block.

The syntax is as follows:

try:

……..

Suspicious code that can raise exception

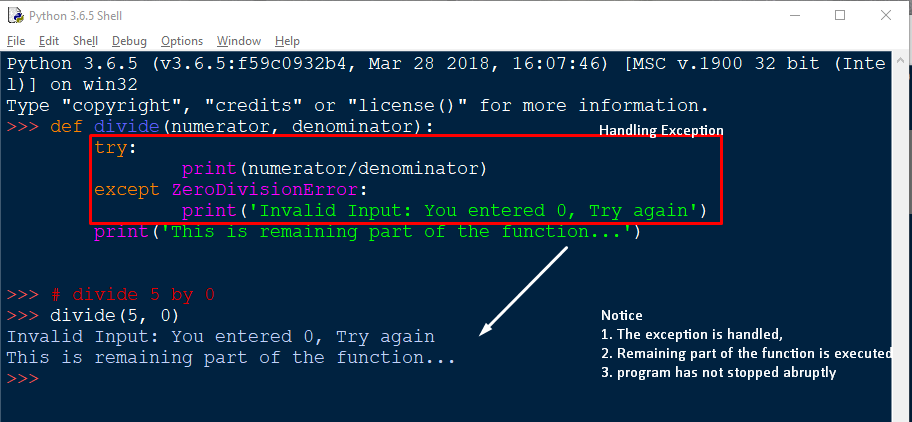
……..

except:

……..

Handling the exception if it occurred

……..

**

**Try:**

The try block is the block where suspicious code ( a code that can cause exceptions) is placed. If you have such statements that may/may not cause an exception and you want to handle it, then such code/statements are placed inside the try block.

try:

statement1 inside the try block

statement2 inside the try block

**Except:**

Except [ExceptionName]

The Except block expects the user to supply the exception name. A **Try block can have multiple except blocks** ( one or more). Each except block is meant to handle the specific exception (which is mentioned by the user ) for instance,

except Exception1:

If there is exception 1, then execute this block to handle

except Exception2:

If there is exception 2, then execute this block to handle

except:

This block handles all types of exceptions

**The Handling mechanism**

* First, the try clause (the statement(s) between the [try](https://docs.python.org/3/reference/compound_stmts.html#try) and [except](https://docs.python.org/3/reference/compound_stmts.html#except) keywords) is executed.
* If no exception occurs, the except clause is skipped and execution of the [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement is finished.
* If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the [except](https://docs.python.org/3/reference/compound_stmts.html#except) keyword, the except clause is executed, and then execution continues after the [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement
* If an exception occurs which does not match the exception named in the except clause, it is passed on to the following/next except clause/outer [try](https://docs.python.org/3/reference/compound_stmts.html#try) statements; if no handler is found, it is an unhandled exception and execution stops with a message as shown above.
* if no specific type of exception is mentioned after the except, then that except clause is executed for all types of exception

**ELSE block:**

* It’s an **optional** clause (You may/may not include it in your exception handling).
* If included, it **must follow all except clauses** in other words must be present after the last except clause
* It **is executed** if the **try block doesn’t raises any exception**. If some exception is raised, the code/lines of code inside this block will never be executed.
* If you want to perform some operation if the exception is not raised But don’t want to handle any accidental exception raised by such code you place it inside the else block.

try:

operation\_that\_can\_throw\_ioerror()

except IOError:

handle\_the\_exception\_somehow()

else:

# we don't want to catch the IOError if it's raised

another\_operation\_that\_can\_throw\_ioerror()

finally:

something\_we\_always\_need\_to\_do()

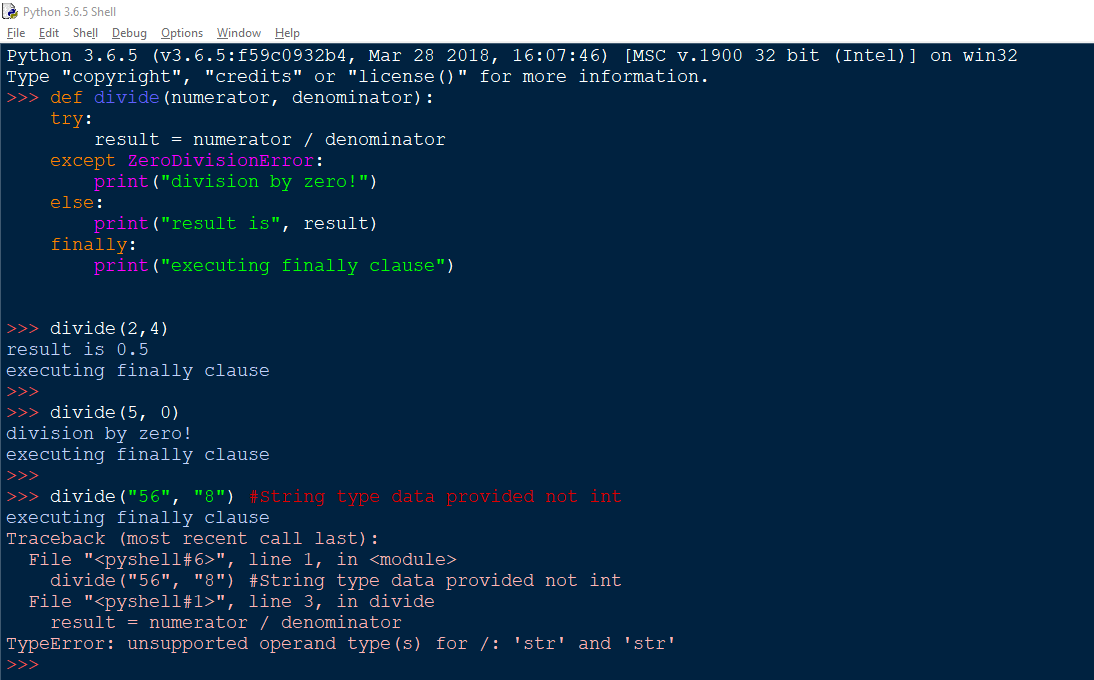
If you just put another\_operation\_that\_can\_throw\_ioerror() after operation\_that\_can\_throw\_ioerror, the except would catch the second call's errors. And if you put it after the whole try block, it'll always be run, and not until after the finally.

You will read about finally block after this section. For now you may note that this block will always be executed regardless of the exception is raised or not. The else lets you make sure

1. the second operation's only run if there's no exception,
2. it's run before the finally block, and
3. any IOErrors it raises ( the code inside it, raises ) aren't caught here

**FINALLY block:**

* It’s an **optional** clause
* This block will always be executed regardless of the exception is raised or not before leaving the try block
* It is intended to define clean-up actions that must be executed under all circumstances.



**Raising Exceptions:**

The [raise](https://docs.python.org/3/reference/simple_stmts.html#raise) statement allows you to manually raise the specified exception to occur.

Syntax:

raise Exception\_to\_be\_raised()

OR  
 raise Exception\_to\_be\_raised

It accepts only one argument, that specifies the type of exception to raise

The argument must be either an exception instance or an exception class

If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments

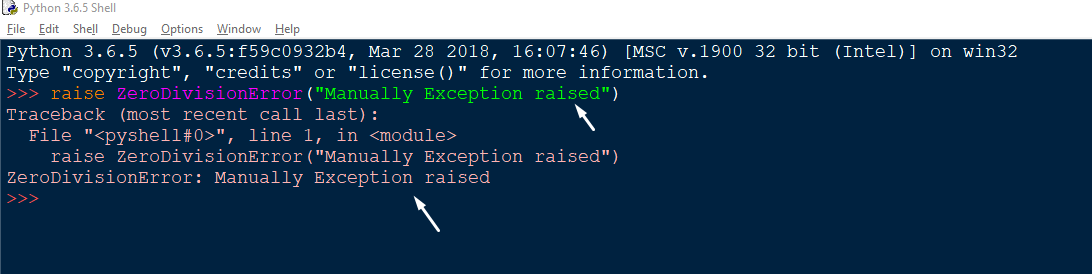
def MyFunction( age ):

if age < 20:

raise "Invalid age!", age

# The code below to this would not be executed

# if we raise the exception



**User- Defined Exceptions:**

* Python also allows you to create your own exceptions i.e. user-defined exceptions. You can do so deriving classes from the standard built-in exceptions
* Exceptions should typically be derived from the [Exception](https://docs.python.org/3/library/exceptions.html#Exception) class, either directly or indirectly
* This is useful when you need to display more specific information when an exception is caught

class MyException(RuntimeError):

def \_\_init\_\_(self, arg1, message):

self.args = arg

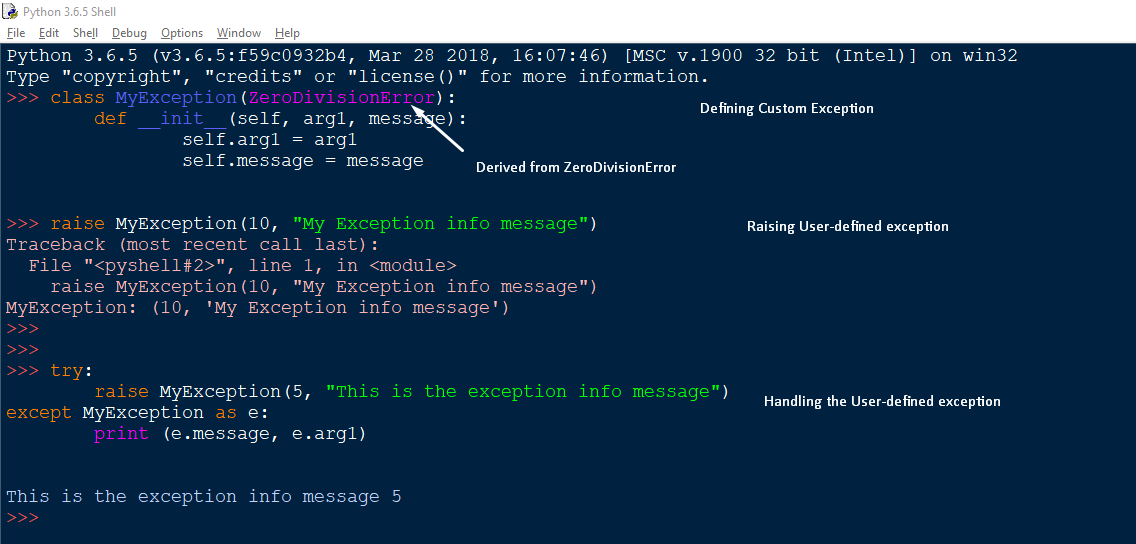
self.message = message

try:

raise MyException("My First exception!")

except MyException as e:

print e.args



**FILE HANDLING IN PYTHON**

Until now in our python programs we have been saving data in variables and data structures like list, dictionaries, tuples etc.

You must have noticed one crucial thing, the data remains saved only for as long as our python program is running, in plain terms, data was saved only locally (RAM to be more precise),

Since RAM is a volatile memory, there is no way to access the data stored once the program has been terminated or it’s execution has been over.

Programming languages uses RAM as the default memory allocation location during program execution

Fortunately, Python does provide a more permanent manner to store and access our data, known as ***FILES***.

When we are working with Python, you don't need to import a library in order to read and write files. Its handled natively in the language.

A **File** is a named location on disk to store related information. It is used to permanently store data in a non-volatile memory (e.g. hard disk), which can be later used to access the data long after our python program has terminated.

In Python the file operations takes place in the following order.

1. Creating/Opening the file
2. Reading or writing contents from/to it (perform operation)
3. Closing the file
4. **Opening a File**

* Python has a built-in function – open() to open files
* open() returns a file object
* It accepts three arguments
  + File
  + Mode
  + Buffering

file object = open(file, mode, buffering)

The filename that we specify here should be the name of the file that exists within the working directory(The folder which has our program) of the python program, otherwise python will throw an error ”No such file or directory

|  |  |
| --- | --- |
| File | the name of the file to open/create |
| mode | specifies the mode to open file in.  Reading, writing, appending etc. |
| Buffering | If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file. If you specify the buffering value as an integer greater than 1, then buffering action is performed with the indicated buffer size |

f = open(“GameOfThrones.txt”) # file present in the working directory

f = open(“C:/Users/Pooja/Desktop/GameOfThrones.txt”) # Full file path

**1.1 modes:**

|  |  |
| --- | --- |
| ‘r’ | Open a file for reading (default) |
| ‘w’ | Open a file for writing. Creates a new file it it does not exists or truncates the file if it exists. |
| ‘a’ | Open for appending at the end of the file without truncating it. Creates a new file if it does not exist. |
| ‘x’ | Open a file for exclusive creation. If the file already exists, the operation fails. |
| ‘t’ | Open the file in text mode. (Default) |
| ‘b’ | Open the file in binary mode. |
| ‘+’ | Open a file for updating (reading & writing) |

The modes can be combined to create a custom desired mode

rb+ - To read & write the data in binary mode

f = open("myFile.txt") # equivalent to 'r' or 'rt'

f = open("myFile.txt",'w') # write in text mode

f = open("myFile.txt",'rb+') # read and write in binary mode

**2.1 Reading a File in Python**

Python provides three flexible methods to read the data from the file object obtained so far. These are

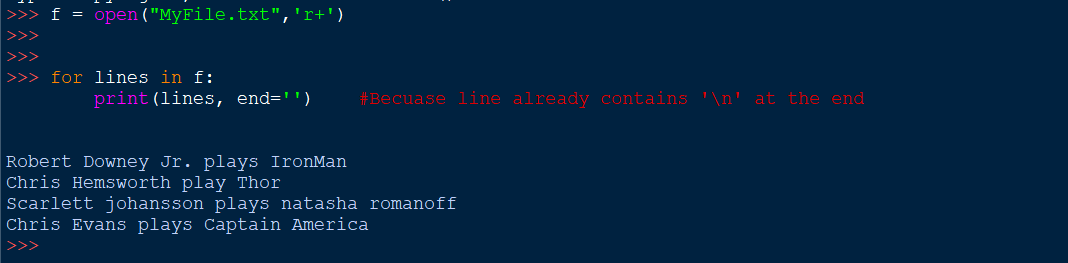
* 1. read(size)
  2. readline( )
  3. readlines( )

**2.1.i) read(size)**

the read(size) method is used to read in size number of data. If size parameter is not specified, it reads and returns up to the end of the file. For instance, in a text file number of data would mean number of characters to read.

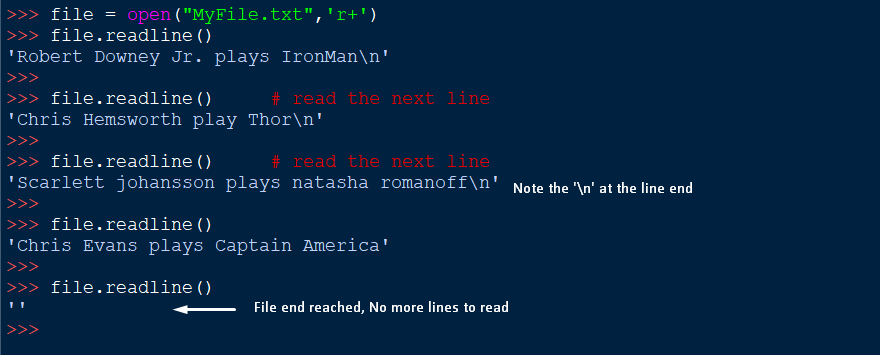


Notice, the new lines are represented as \n characters. We can use a for loop and create a workaround by making use of the end



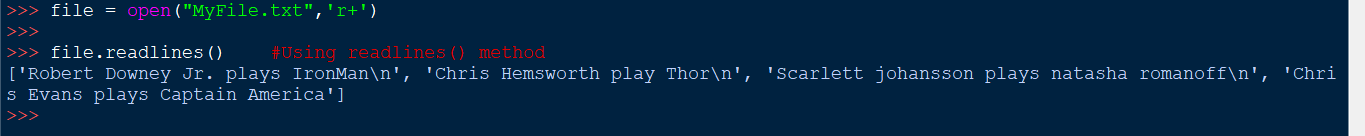
**2.1.ii) readline()**

The python readline() methods reads one line at a time from the file including the newline character (‘\n’)



**2.1.iii) readlines()**

The readlines() methods reads all the lines from the file one by one(sequentially) until the EndOfFile EOF is reached.



**2.2 Writing into Files**

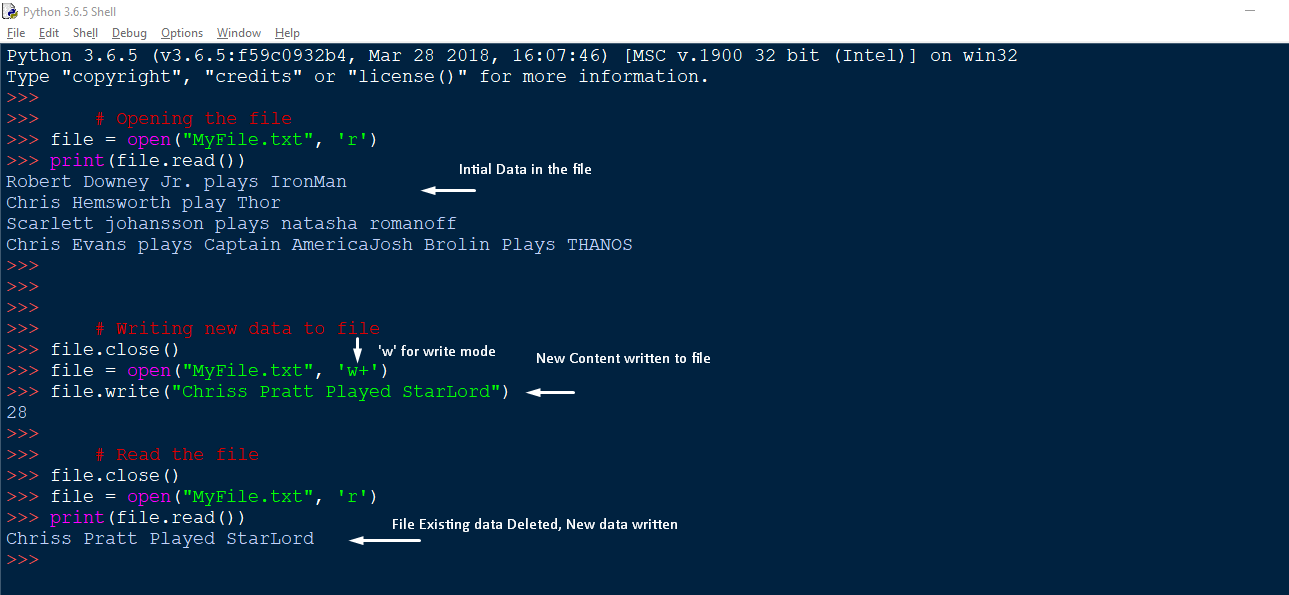
* To write data into the file we use the python write() function
* It returns the number of characters written to the file
* CAUTION: To write data to a file, the file must be open in either

1. write ‘w’,
2. append ‘a’,
3. exclusive creation ‘x’

* using ‘w’ to write the file will overwrite the file if it already exists, in other words, it will delete all the previous data.

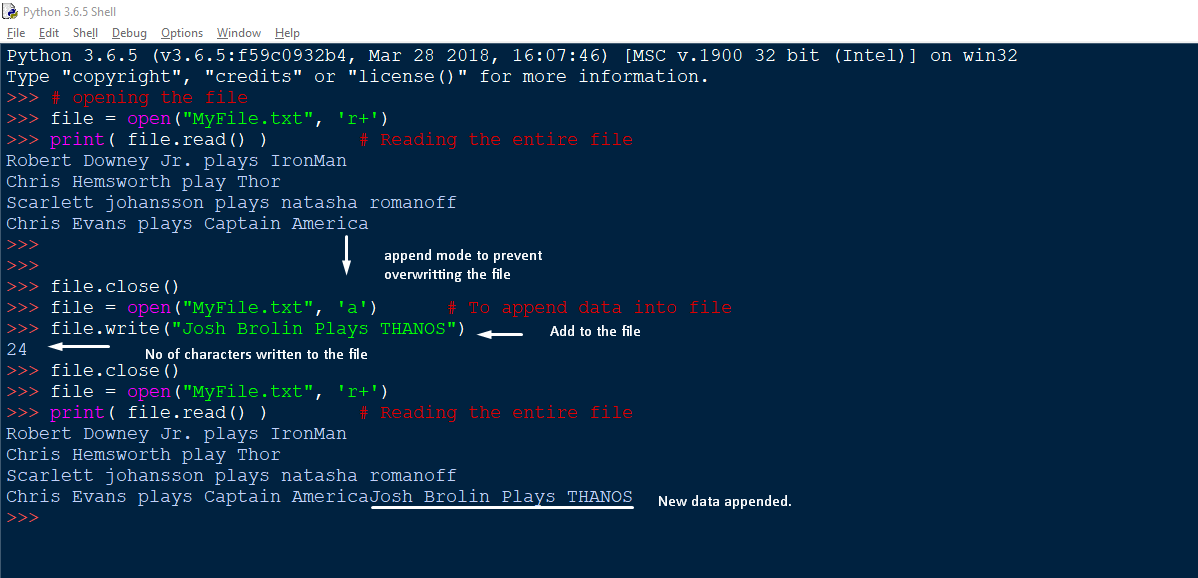
**2.2.i) Write mode ‘w’**

* This modes overwrites the file
* The original/old file data is completed deleted
* Only new data is present in the file after the operation



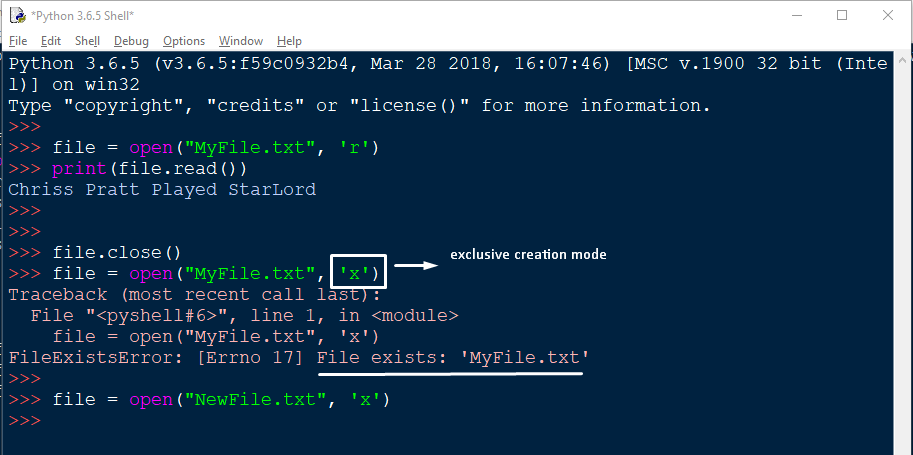
**2.2.ii) Append mode ‘a’**

* + This modes adds the new contents to the end of the files.
  + Original data is preserved
  + New data is appended



**2.2.iii) Exclusive Creation mode ‘x’**

* + This opens files for exclusive creation.
  + If the file already exists an error occurs.
  + If the file doesn’t exists already, a new file is created



1. **Closing the File**

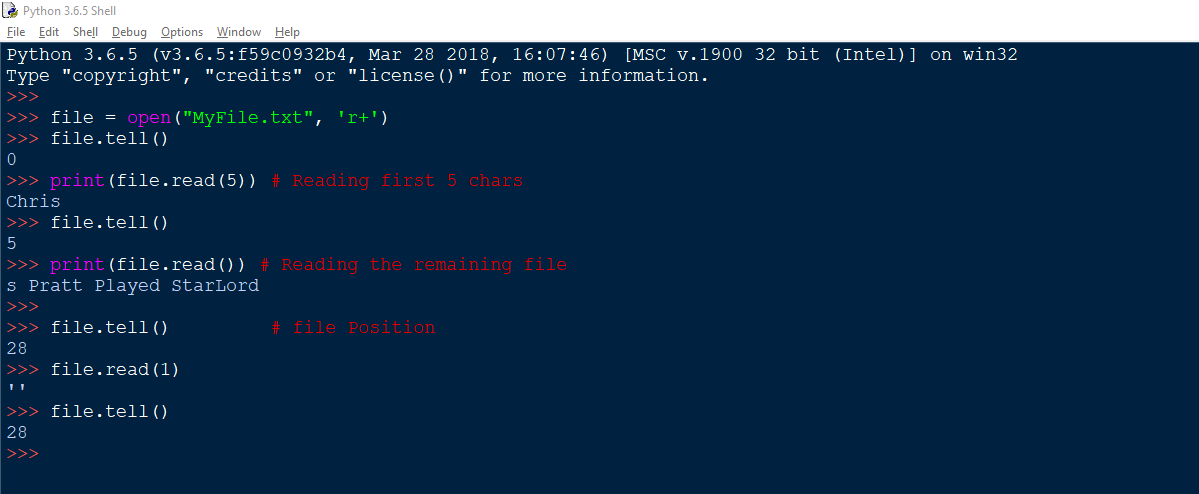
* When we have completed our file operation, we need to close our file
* The close() function is used to close the file
* On calling the close() function, the resources that were tied to file will be freed up.
* One must not rely on garbage cleaner to close the file and explicitly do so.

file.close()

1. **File Read positions**

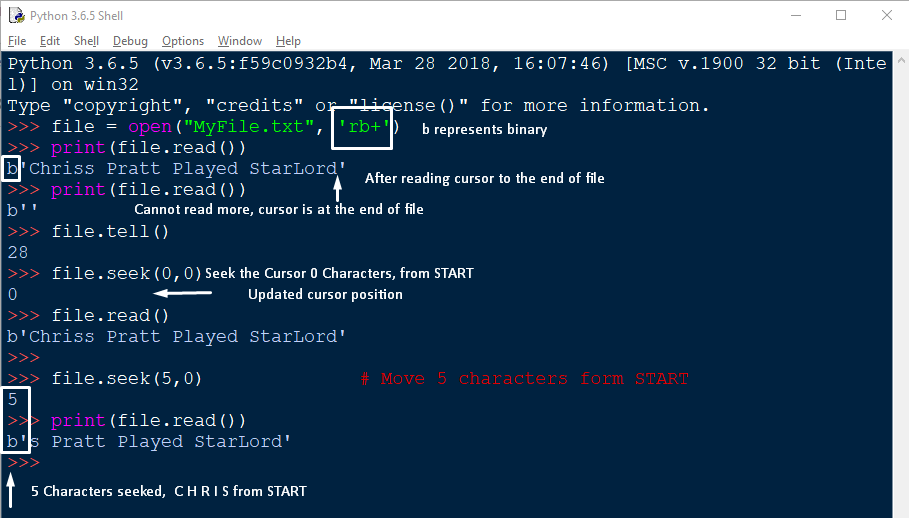
**3.1)** **tell()**

* This method returns the current location in the file

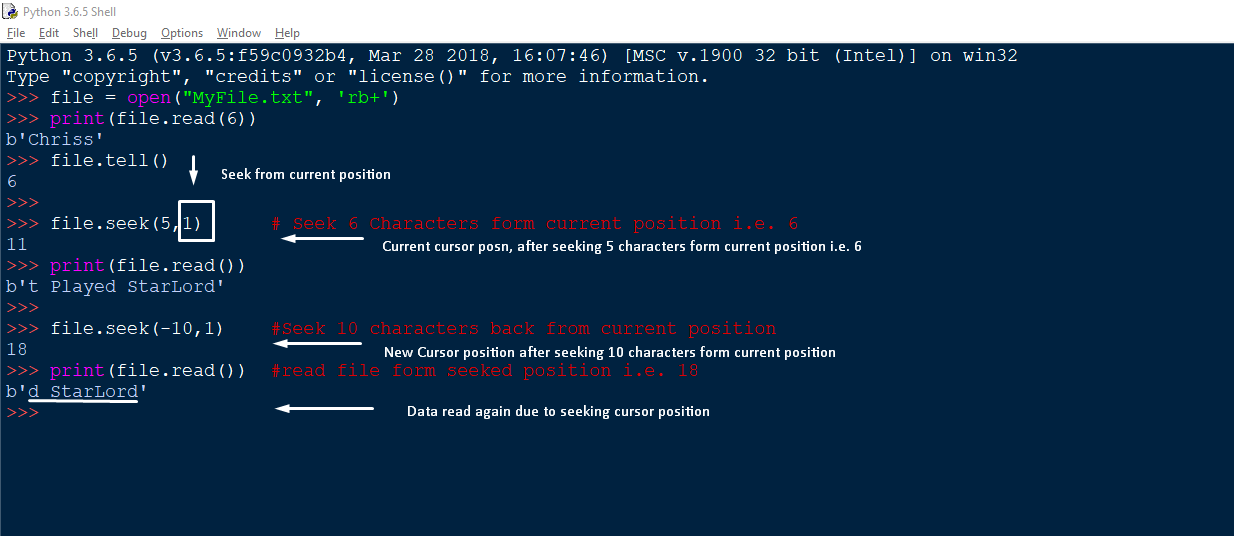


**3.2) seek(offset, from=SEEK\_SET)**

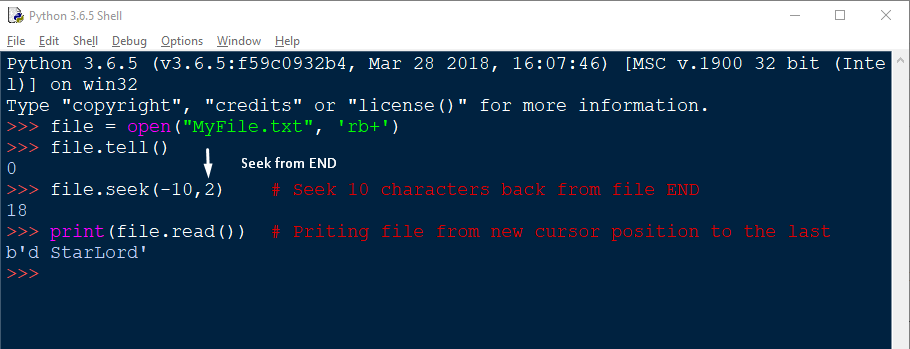
* The seek function changes the cursor position in the file.
* It seeks backwards or forwards
* The offset determines the no. of characters to move say, int value
* The form specifies the seek operation form 3 position
  + 0 – Start of the file
  + 1 – Current Position
  + 2 – End of the file

From start: (offset, 0)

Form Current: seek(offset, 1)



From End: seek(offset, 2)



1. **Opening the files using ‘with’ statements**

You can also work with file objects using the with statement. It is designed to provide much cleaner syntax and exceptions handling when you are working with code. That explains why it's good practice to use the with statement where applicable.

One bonus of using this method is that any files opened will be closed automatically after you are done. This leaves less to worry about during cleanup.

To use the with statement to open a file:

with open(“MyFile.txt”) as file:

for lines in file:

print(lines, end=‘ ’)

The file object f is only valid inside the with block and the file closed automatically when we come out of the with block.

1. **Renaming & Deleting the files**

While you were using the read/write functions, you may also need to rename/delete a file in Python. So, there comes a os module in Python which brings the support of file rename/delete operations

**6.1) Rename:**

* To rename a file we need to import the os module and use the rename method provided in it

rename(cur\_file\_name, New\_file\_name)

* The rename() methods expects two arguments, the current filename and the new filename

Example:

import os

# Rename MyFile.txt 🡪 One.txt

os.rename(“MyFile.txt”,”One.txt”)

**6.2) Remove:**

* In Python the os module provides the method remove(), that can be used to delete files in python
* The function excepts only a single argument, the filename, to be deleted.

remove(filename)

example:

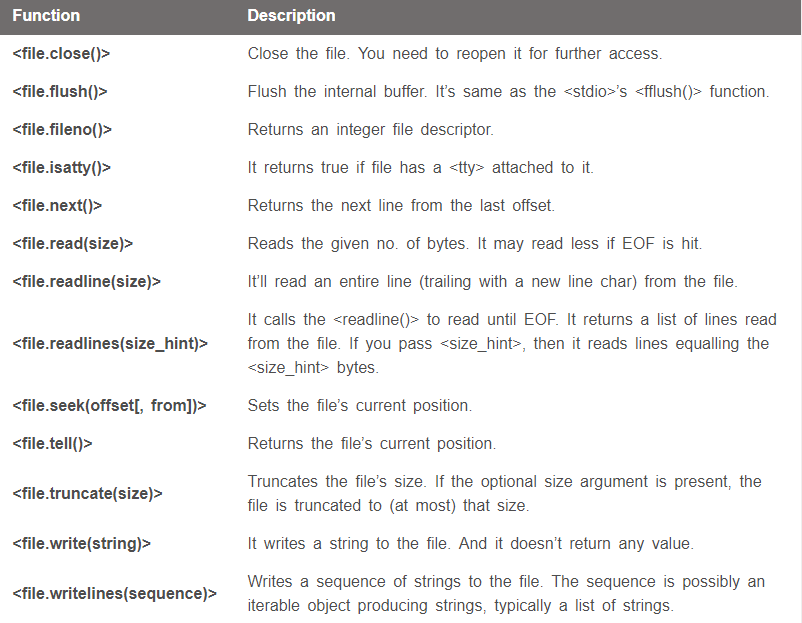
import os

# To delete MyFiles.txt

os.remove(“MyFiles.txt”)

1. **Python File methods**

|  |  |
| --- | --- |
| close() | Close an open file. It has no effect if the file is already closed. |
| detach() | Separate the underlying binary buffer from the TextIOBase and return it |
| fileno() | Return an integer number (file descriptor) of the file |
| flush() | Flush the write buffer of the file stream. |
| isatty() | Return True if the file stream is interactive. |
| read(n) | Read atmost n characters form the file. Reads till end of file if it is negative or None. |
| readable( | Returns True if the file stream can be read from. |
| readline(n=-1) | Read and return one line from the file. Reads in at most n bytes if specified. |
| readlines(n=-1) | Read and return a list of lines from the file. Reads in at most n bytes/characters if specified. |
| seek(offset,from=SEEK\_SET) | Change the file position to offset bytes, in reference to from (start, current, end). |
| seekable() | Returns True if the file stream supports random access. |
| tell() | Returns the current file location. |
| truncate(size=None) | Resize the file stream to size bytes. If size is not specified, resize to current location. |
| writable() | Returns True if the file stream can be written to. |
| write(s) | Write string s to the file and return the number of characters written. |
| writelines(lines) | Write a list of lines to the file. |

****